Palo Alto Research Centers

Cryptographic Sealing for Information Secrecy and Authentication

David K. Gifford

XEROX

Cryptographic Sealing for Information Secrecy and Authentication

by David K. Gifford

CSL-82-1 FEBRUARY 1982

© Xerox Corporation 1982. All rights reserved.

Abstract: A new protection mechanism is described that provides general primitives for protection and authentication. The mechanism is based on the idea of sealing an object with a key. Sealed objects are self-authenticating, and in the absence of an appropriate set of keys, only provide information about the size of their contents. New keys can be freely created at any time, and keys can also be derived from existing keys with operators that include *Key-And* and *Key-Or*. This flexibility allows the protection mechanism to implement common protection mechanisms such as capabilities, access control lists, and information flow control. The mechanism is enforced with a synthesis of conventional cryptography, public-key cryptography, and a threshold scheme.

A version of this paper will appear in *Communications of the ACM*, vol. 25 no. 4, April 1982.

CR Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—access controls, authentication, cryptographic controls, information flow controls; E.3 [**Data**]: Data Encryption—public-key cryptosystems.

General Terms: Security

Additional Key Words and Phrases: cryptographic sealing, seal, unseal, key, secrecy, conventional cryptosystems

XEROX

Xerox Corporation
Palo Alto Research Centers
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

In order to trust computers with sensitive information it is necessary to take steps to ensure that information stored in them will only be disclosed to authorized users. Personal information is now routinely stored in computers, and thus computer security is a necessary precondition for personal privacy. In addition, it is widely recognized that information can be sold as a product. In this case, it is important to be able to restrict information access to paying customers.

The problem of computer security can be considered to consist of four distinct components: *secrecy* (ensuring that information is only disclosed to authorized users), *authentication* (ensuring that information is not forged), *integrity* (ensuring that information is not destroyed), and *availability* (ensuring that access to information can not be maliciously interrupted).

This paper describes a new protection mechanism called *cryptographic sealing* that provides primitives for secrecy and authentication. The mechanism is enforced with a synthesis of conventional cryptography, public-key cryptography, and a threshold scheme.

The new mechanism is based on the idea of *sealing* an object with a *key*. *Sealed objects* are self-authenticating, and in the absence of an appropriate set of keys, only provide information about the size of their contents. Thus, keys are the basic unit of protection and authentication in the mechanism. New keys can be freely created at any time, and keys can also be derived from existing keys with operators that include *Key-Or* and *Key-And*. These operators allow protection structures to be established that allow any member of a set of keys to unseal an object, that require every member of a set of keys to unseal an object, or any combination of these extremes. This flexibility allows cryptographic sealing to implement common protection mechanisms such as capabilities, access control lists, and information flow control.

Objects and sealed objects are simply arrays of bytes. In order to update the value of a sealed object it is necessary to unseal it, change its value, and reseal it.

Programs that use the protection mechanism (hereafter called *clients*) must operate in a secure environment so they can safely manipulate unsealed objects. A simple way of providing such an environment would be for each client to execute on a separate physical processor. For the purposes of this discussion we will assume that every client executes on a *secure processor* that is protected from every other client.

Our description of the protection mechanism and its applications is organized into seven sections. Section 2 discusses the general framework of the mechanism, and relates it previous work. Section 3 informally and formally describes a model of the protection mechanism, and shows how the mechanism can be used to implement common protection structures. Section 4 discusses in detail the algorithm that is used to implement the mechanism, and the extent to which the mechanism can be trusted. Section 4 can be skipped if the reader is primarily interested in what the mechanism can do, as opposed to how it works. Section 5 examines some practical considerations. Section 6 is a comparative analysis of cryptographic sealing and traditional protection mechanisms.

The algorithms in the paper are described in a dialect of Lisp 1.5 [McCarthy et al. 62] for pedagogical reasons. Lisp lends itself to the transformation of objects to byte strings and back again (see the descriptions of *Encode* and *Decode* in the appendix), and it allows sets of objects to be

represented as lists. The appendix should provide enough information to enable the reader to understand the code shown below.

2. Framework

In order to compare the present work with previous protection systems [Saltzer and Schroeder 79] we need to provide an appropriate framework. We will call a protection mechanism *active* if it is placed between a client and protected information, as shown in Figure 1. An active protection mechanism serves to inhibit unauthorized client requests to storage. If a client could bypass an active protection mechanism, it could gain unauthorized access to protected information. Thus, active protection mechanisms depend on a *security envelope* that a client can not penetrate. As shown in the figure, system administrators and operators are inside of this envelope, because they have direct physical access to the system.

The protection mechanism this paper introduces is the first example of a general purpose *passive* protection mechanism. It is passive because there are no restrictions placed on a client's access to storage. However, a client is only able to decipher information that it is authorized to see. As shown in Figure 2, a passive protection system operates as part of the client.

A passive protection system can only address the secrecy and authentication aspects of computer security. Section 3.2 describes how an active protection mechanism can be used to supplement a passive system to provide integrity and availability. When passive and active systems are used together in this manner a failure of the active protection system only effects integrity and availability. Secrecy and authentication are still guaranteed by the passive protection system. In a similar way, if the ability to revoke privileges is desired, an active protection system would be used as described in Section 5.

Some of the ideas in this paper have appeared in other forms. Morris [Morris 73] discussed the concept of sealing objects, but he did not present a way to create general protection structures, and his mechanism was not enforced by cryptography. Chaum and Fabry [Chaum and Fabry 78], Lindsay and Gligor [Lindsay and Gligor 78], and Needham [Needham 79] independently observed that cryptography can be used to authenticate objects such as capabilities. Gudes [Gudes 80] described a way to use cryptography to implement a form of access control lists without groups or indirect keys.

3. The Protection Mechanism

To introduce the protection mechanism we first describe a model of the system in formal and informal terms, and then we show how the mechanism can be used to implement capabilities, access control lists, and information flow control.

3.1 Model

As we have mentioned, keys are the basic unit of secrecy and authentication in the protection mechanism. There are two ways to generate a key. It is possible to generate a brand new random

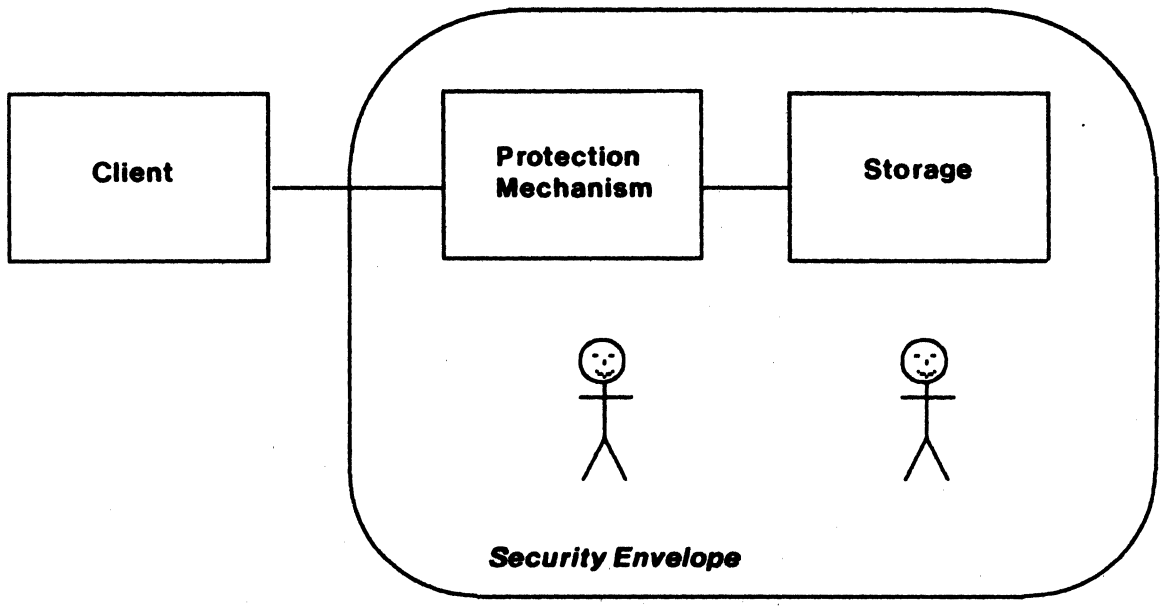


Figure 1. An Active Protection Mechanism

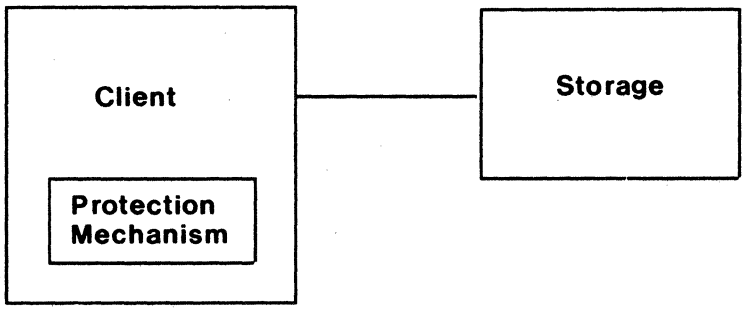


Figure 2. A Passive Protection Mechanism

key that does not depend on any other keys; such a key is called a *base key.* It is also possible to generate a key that is a function of existing keys; such a key is called a *derived key.* Derived keys are used to implement protection structures that can not be realized with base keys alone.

The keys required to unseal an object depend on the structure of the key used to seal the object. For example, to unseal an object that was sealed with *Key-Or[ka, Key-And[kb, kc]]* one needs either *ka* or both *kb* and *kc.* To be as explicit as possible, we introduce the *unseals* relation between a set of keys and a key. As we shall see in a moment, if a set of keys S unseals key K, then S can be used to unseal any object that has been sealed with K. If key set S unseals K we will write $S \gg K$. If $S \gg K$ we also say that *K is unsealed by S.*

Seal and *Unseal* are the primitive functions of the protection mechanism. *Seal* seals an object with a key, and *Unseal* reverses the sealing process. The type *Key* is used below to denote either a base or a derived key. ("/" is used to separate input and result arguments; see the appendix for further details).

Seal[x: Any, k: Key / sx: Sealed-Object]

> *Seal* seals x with k, returning a sealed object. *Seal* operates by value; x and k are not modified. *Seal[x, NIL]* is x.

Unseal[sx: Sealed-Object, ks: List[Key] / x: Any]

> *Unseal* unseals *sx* with the set of keys contained in the key set *ks.* If *Unseal* is successful, it returns x (the original input value to *Seal*). Otherwise *Unseal* returns *'Failed.* Unseal does not modify *sx.*

The protection mechanism provides two properties. The *secrecy property* states that a sealed object is useless to someone who does not have a set of keys that unseals the key that was used to seal the object. The *authentication property* states that *Unseal* will only return values that were in fact properly sealed.

> **Secrecy.** x can be recovered from *Seal[x, k]* with a set of keys S if and only if $S \gg k$.
>
> **Authentication.** If x was not sealed with a key that is unsealed by S then the result of *Unseal[x, S]* will be *'Failed.*

The following functions create keys, and completely define the unseals relation.

Create-Base-Key[/ k: Key]

> *Create-Base-Key* creates a new regular base key. Regular base keys are the simplest kind of keys. To unseal *Seal[x, k]* requires k. In other words, a key set unseals k if and only if it contains k.
>
> $$(S \gg k) \equiv (k \in S)$$

Key-Pair ← Record[keya: Key, keyb: Key]

Create-Key-Pair[/ kp: Key-Pair]

Create-Key-Pair creates a pair of base keys. These keys are related to each other in the following way. To unseal *Seal[x, keya]* requires *keyb*, and to unseal *Seal[x, keyb]* requires *keya*. In other words, a key set unseals *keya* if and only if it contains *keyb*, and a key set unseals *keyb* if and only if it contains *keya*.

$$(S \gg keya) \equiv (keyb \in S)$$
$$(S \gg keyb) \equiv (keya \in S)$$

Key-And[ka: Key, kb: Key / dk: Key]

Key-And creates a derived key that is the logical and of *ka* and *kb*. To unseal *Seal[x, dk]* requires either *dk* or a set of keys that will unseal both *ka* and *kb*. That is, a key set unseals *dk* if and only if it unseals both *ka* and *kb* or it includes *dk*.

$$(S \gg dk) \equiv ((S \gg ka) \wedge (S \gg kb)) \vee (dk \in S)$$

Key-Or[ka: Key, kb: Key / dk: Key]

Key-Or creates a derived key that is the logical or of *ka* and *kb*. To unseal *Seal[x, dk]* requires either *dk* or a set of keys that unseals *ka* or *kb*. That is, a key set unseals *dk* if and only if it unseals *ka* or *kb* or it includes *dk*.

$$(S \gg dk) \equiv (S \gg ka) \vee (S \gg kb) \vee (dk \in S)$$

Key-Quorum[q: Integer, key-list: List[Key] / dk: Key]

Key-Quorum creates a derived key. *key-list* is a list of an arbitrary number of keys. To unseal *Seal[x, dk]* requires either *dk* or a set of keys that unseals *q* distinct keys from *key-list*. In other words, a set of keys unseals *dk* if and only if it includes *dk* or if it unseals *q* distinct keys from *key-list*. Note that *Key-Quorum* can be used to implement *Key-And* and *Key-Or* (with $q=2$ and $q=1$ respectively), but as we shall see the implementations of *Key-And* and *Key-Or* are more efficient than *Key-Quorum*. For some *q* combination $k_1, ..., k_q$ drawn from *key-list*:

$$(S \gg dk) \equiv (\wedge_{1 \leq i \leq q} S \gg k_i) \vee (dk \in S)$$

Submaster[k: Key / dk: Key]

Submaster creates a derived key. To unseal *Seal[x, dk]* requires either *dk* or a set of keys that unseals *k*. In other words, a key set unseals *dk* if it unseals *k* or if it includes *dk*.

$$(S \gg dk) \equiv (S \gg k) \vee (dk \in S)$$

Seal-Only[k: Key / dk: Key]

Seal-Only creates a derived key. To unseal *Seal[x, dk]* requires a set of keys that unseals *k*. In other words, a key set unseals *dk* if and only if it unseals *k*. A key set that includes *dk*

is not sufficient to unseal *dk*.

$$(S >> dk) \equiv (S >> k)$$

Create-Indirect-Key[k: Key / ik: Indirect-Key]

Create-Indirect-Key creates an indirect key. To unseal *Seal[x, ik]* either requires *ik* or a set of keys that unseals *k*. In other words, a key set will unseal *ik* if it contains *ik* or if it unseals *k*. Once an indirect key has been created, it can be changed with *Change-Indirect-Key*. Because indirect keys can be altered, the following statement is an implication.

$$(S >> k) \vee (ik \in S) \rightarrow (S >> ik)$$

Change-Indirect-Key[ik: Indirect-Key, nk: Key]

Change-Indirect-Key changes an indirect key. After Change-Indirect-Key has been performed, to unseal an object that has been sealed with *ik* either requires *ik* or a set of keys that unseals *nk*. In other words, *ik* is changed such that if a key set unseals *nk* it now also unseals *ik*. *Change-Indirect-Key* does not provide revocation as discussed in Section 5.

$$(S >> nk) \rightarrow (S >> ik)$$

Let us consider an example to make the idea of an indirect key clear. Imagine that the following occurs:

```
kevin ← Create-Base-Key[];
ik ← Create-Indirect-Key[kevin];
sx ← Seal[x, ik];
```

At this point *sx* can be unsealed with either *ik* or *kevin*. The indirect key is then updated:

```
harry ← Create-Base-Key[];
Change-Indirect-Key[ik, Key-Or[kevin, harry]];
```

After the Change-Indirect-Key occurs, *ik*, *harry*, or *kevin* are required to unseal *sx*.

If $S >> k1$ and $S >> k2$, then *Unseal[Seal[Seal[x, k1], k2], S]* will be *Seal[x, k1]*. In order to recover an object that may have been sealed many times *RUnseal* can be used.

RUnseal[sx: Sealed-Object, ks: List[Key] / x: Any]

RUnseal applies *Unseal* as many times as necessary until its result is not sealed. *RUnseal* always applies *Unseal* at least once.

RUnseal-List[sl: List[Sealed-Object], ks: List[Key] / lx: List[Any]]

RUnseal-List applies *RUnseal* to a list of sealed objects, and returns a list of unsealed objects. If *ks* does not unseal an element of *sl*, the corresponding element of *lx* will be *'Failed*.

3.2 Applications

It is possible to use the protection primitives we have described to implement a variety of other protection mechanisms, including capabilities, access control lists, and information flow control. For the purposes of this discussion we will consider a decentralized system, constructed of a number of file servers and clients.

Assume each user of the system has a personal list of keys, or *key ring*, that defines his privileges. When a key ring is stored, it is sealed with a key that only its owner knows. A user authenticates himself to the system by providing the key to his key ring. The key ring key is used to unseal the user's key ring in his secure processor, resulting in his list of privileges.

To provide integrity and availability we introduce a simple active protection mechanism. Imagine that each object is assigned a unique set of passwords, one for each of its independent privileges. We will call these passwords *guards*. Because each object has a unique set of guards, they must be stored with the object. For example, suppose file F is assigned write guard G. The file server that stores F would require that G be presented for each write access.

We will further assume that objects are named with variable size names called *references*. A reference always includes a unique identifier for its referent, and may include other fields as well.

Capabilities

A capability is an unforgable ticket which permits a possessor to access the object it names with certain privileges [Dennis and Van Horn 66]. No special privileges are required in our system to make a copy of a capability.

Capabilities can be implemented by including a set of keys and guards in an object's reference. For example, imagine object O is sealed with a key that can be unsealed by K, and G is the write guard for object O. If a reference for O includes K, then it can be used to read O. If the reference also includes G, it can be used to read or write O. Without G or K it is impossible to read or write O. To the extent that keys and guards are considered impossible to guess, capabilities can be considered unforgable. A capability reference is shown in Figure 3.

Access Control Lists

An access control list system associates a list of users with each object. This list describes who may access the object, and with what privileges [Saltzer and Schroeder 79].

To implement access control lists each user of the system creates a key pair, and makes one of the keys of this pair public. Section 5 discusses how one user can reliably learn another user's public key. A user keeps the private half of his key pair on his key ring. Thus if user X seals an object with user Y's public key, only user Y will be able to unseal it. This result follows directly from the public-key cryptography that is used to implement key pairs.

Access control lists are implemented by sealing the privileges in a capability reference, as shown in Figure 4. In general, a key or guard is sealed with the *Key-Or* of the users' keys that have been granted the corresponding privilege. Once sealed, these references can be placed in a public directory system. Only users that have been granted a privilege will be able to unseal its corresponding key or guard.

It is often desirable to be able to grant privileges to a group of users and allow the members of

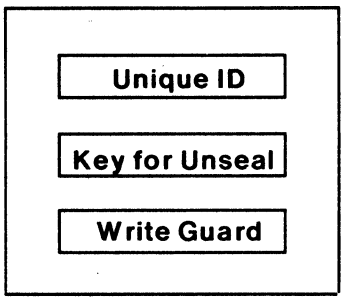
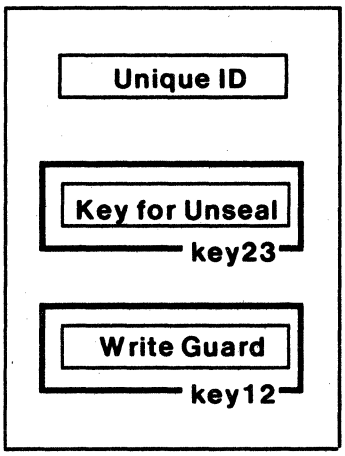


Figure 3. Object Reference: Capability



Read
<Jones>, <Smith>, <Zephyr>

Write
<Jones>, <Smith>

Key23 = Key-Or[<Jones>, <Smith>, <Zephyr>]

Key12 = Key-Or[<Jones>, <Smith>]

Figure 4. Object Reference: Access Control List

the group to change over time. If revocation is not required, indirect keys can be used to define a group as

Group-Key ← Seal-Only[Create-Indirect-Key[
Key-Or[u_1 ... Key-Or[u_{n-1}, u_n]]]];

where u_1 ... u_n are the users' keys that are members of the group. The members of the group can be altered with *Change-Indirect-Key.*

Information Flow Control

In an information flow control scheme [Denning 76] each object is labeled with one or more classifications, and the output of a computation is labeled with the union of the classifications of its inputs. An example of an information flow control scheme is the military system of classification. If *Top Secret* and *Crypto* information are used in a report, then the report would be classified *Top Secret, Crypto.*

Information flow control schemes are usually nondiscretionary, meaning that the classification of a computation's output is fixed, and can not be altered by a client. In the information flow control scheme we present a client chooses the classifications of its outputs. Once a classification is specified, the protection mechanism ensures that it is properly enforced.

Cryptographic sealing can be used to implement information flow control in the following manner. Represent each classification by a key. If *O* is created from objects that have classifications C_1 ... C_n, seal the privileges in *O's* reference with *Key-And[*C_1 ... *Key-And[*C_{n-1}, C_n*]].*

For example, imagine that the fictitious company Sierra has two divisions, a medical division and an office division. Sierra would like to enforce the policy that information that is private to one division is only accessible to employees of that division. Furthermore, Sierra has financial information that only senior managers are allowed to access. Sierra could create keys to represent these classifications as follows:

Medical ← Seal-Only[Create-Indirect-Key[
Key-Or[<adams>, ... Key-Or[<thatch>, <west>]]]];

Office ← Seal-Only[Create-Indirect-Key[
Key-Or[<jones>, ... Key-Or[<rainbow>, <smith>]]]];

Financial ← Seal-Only[Create-Indirect-Key[
Key-Or[<adams>, ... Key-Or[<irby>, <welch>]]]];

The names in brackets represent the public keys of people that are members of each classification. Indirect keys are used so that membership of each classification group can be altered later. Information about the overall performance of Sierra could be sealed with *Financial.* Information about the financial performance of the medical division could be sealed with *Key-And[Financial, Medical].*

It is of course possible to combine access control lists and information flow control in a single system. This hybrid structure results when "the need to know" is added to an information flow control policy.

4. Algorithm

The workings of the protection mechanism are described in the following three sections. First, we introduce the expected environment of the protection mechanism. Second, we show how the mechanism works. Third, we discuss the extent to which the mechanism can be trusted.

4.1 Environment

We describe the building blocks of the protection mechanism in detail because with a thorough knowledge of these facilities the reader will find it easier to understand the sections that follow. The five facilities that the protection mechanism uses are cryptography, a threshold scheme, checksums, indirection, and unique identifiers. It is difficult to motivate all of these facilities in the abstract, and thus we ask the reader to be patient until the following section where it will become clear why they were introduced. Our treatment of each of these is necessarily brief; for more detail the reader is referred to Gifford [Gifford 81].

4.1.1 Cryptography

Cryptography is used to *encrypt* information to be protected, also called *cleartext*, into *ciphertext*. Encryption is a transformation from the space of possible cleartexts to the space of possible ciphertexts. The transformation selected depends on the key supplied to the encryption function. To *decrypt*, or recover the cleartext of a ciphertext, requires a specific key so the encryption transformation can be inverted.

A cryptographic system is called *perfectly secure* if all that can be discerned from a ciphertext is that the ciphertext exists, and that the corresponding cleartext is not longer than the ciphertext. Perfect security can only be achieved when the entropy of a key is at least as great as the entropy of information encrypted [Shannon 49]. Thus, perfectly secure systems are usually not practical because a key must be at least as long as the cleartext it is used to encrypt.

A cryptographic system is called *computationally secure* if even when enough information is theoretically available to break the system, the amount of computation required to do so is unreasonable. Unfortunately, proving facts about the computational complexity of many interesting algorithms is beyond the reach of current theory. Thus, most practical cryptosystems can not be proven to be computationally secure. In practice, the best that can be done is to invest a substantial amount of effort in trying to think of a way to break a system; if the system survives such an attack, then it is considered to be secure for practical purposes. As unsatisfying as this may be, it is the current state of the art.

The goal of a cryptanalytic attack is to find the key that was used to encrypt a ciphertext. Attacks are classified by the information an intruder has to aid him. It is assumed that an intruder knows the workings of the cryptosystem he is attacking. As the name implies, in a *ciphertext-only attack* an intruder only has ciphertext. In a *known-cleartext attack* an intruder has both the ciphertext and the cleartext that was used to generate it. In a *chosen-cleartext attack* an intruder can choose text to be encrypted and observe the resulting ciphertext. Experience has shown that a cryptosystem that is known to be vulnerable to a chosen-cleartext attack should not be used.

Two generic types of cryptosystems are used to implement cryptographic sealing. As we

introduce each cryptosystem we will provide enough background so the reader can understand its properties. For more information about cryptography see Diffie and Hellman [Diffie and Hellman 79].

The framework of the protection system is very general, and it is designed to accommodate new cryptosystems as they are discovered. The interfaces to the cryptosystems are described in general terms, allowing stronger implementations of the systems to be adopted as they become available.

The two types of keys that have been described as part of the protection model, regular base keys and keys that are half of a key pair, should not be confused with the keys introduced in this section. These two new types of keys, conventional cryptographic keys and public-key cryptographic keys, are used to implement the keys that are part of the model.

Conventional Cryptography

In a *conventional cryptosystem* a cleartext is encrypted with a key, and the same key is used to decrypt the resulting ciphertext. The following functions implement conventional cryptography.

Create-Conventional-Key[/key: Byte-Array]

> Create-Conventional-Key returns a random key that can be used in conjunction with the conventional encrypt and decrypt functions. All of our key creation functions are based on a true random bit generator.

Conventional-Encrypt[clear: Byte-Array, key: Byte-Array / cipher: Byte-Array]

> Conventional-Encrypt encrypts *clear* with *key*, producing *cipher*.

Conventional-Decrypt[cipher: Byte-Array, key: Byte-Array / clear: Byte-Array]

> Conventional-Decrypt decrypts *cipher* with *key*, producing *clear*. If the key specified was not used to encrypt *cipher*, the value of *clear* is undefined.

Public-Key Cryptography

In a *public-key cryptosystem* the key generation function returns a pair of keys. Call the keys of such a pair *keya* and *keyb*. Only *keyb* can decrypt ciphertext enciphered with *keya*, and only *keya* can decrypt ciphertext enciphered with *keyb*. This is in contrast to a conventional cryptosystem, where a single key is used for both encryption and decryption.

Public-key cryptosystems were first suggested in the open literature by Diffie and Hellman [Diffie and Hellman 76]. The name public-key resulted from the observation that with such a system certain keys could be made public, solving in part the problem of key distribution. Public-key cryptosystems have also been called asymmetric cryptosystems.

Rivest, Shamir, and Adleman [Rivest et al. 78] described the first practical implementation of a public-key system. The only cryptanalytic approaches currently known for breaking their scheme are at least as computationally complex as factoring extremely large numbers.

The exact semantics we chose for our definition of public-key cryptography were motivated by the Rivest, Shamir, and Adleman proposal. Some public-key cryptosystems allow ciphertext enciphered with *keya* to be decrypted with *keyb*, but not the converse. In this case we will assume

that two key pairs from such a system are used to implement one of our public-key pairs to provide the appropriate semantics.

The following functions implement public-key cryptography.

PK-Pair ← Record[keya: Byte-Array, keyb: Byte-Array];

Create-PK-Pair[/pair: PK-Pair]

Create-PK-Pair computes a pair of public keys. The keys are random, in the sense that they are a function of a true random number.

PK-Encrypt[clear: Byte-Array, key: Byte-Array / cipher: Byte-Array]

PK-Encrypt encrypts *clear* with one of the members of a key pair, and it returns *cipher.*

PK-Decrypt[cipher: Byte-Array, key: Byte-Array / clear: Byte-Array]

PK-Decrypt decrypts *cipher* with *key*, producing *clear.* If the key specified is not correct, the value of *clear* is undefined.

4.1.2 Threshold Scheme

A *threshold scheme* allows a datum D to be divided into n pieces, such that any k pieces are sufficient to reconstruct D but complete knowledge of any $k-1$ pieces reveals no information about D [Blakley 79, Shamir 79]. A practical implementation of this system was first demonstrated by [Shamir 79].

The following functions implement a threshold scheme.

Threshold-Pieces ← Record[pieces: List]

Threshold-Split[clear: Byte-Array, n: Integer, k: Integer / piece-list: Threshold-Pieces]

Threshold-Split takes *clear* and logically splits it into n pieces, any k of which are sufficient to recover *clear.* These pieces are returned as an n element list.

Threshold-Recover[piece-list: Threshold-Pieces / clear: Byte-Array]

Threshold-Recover takes a list of pieces, and if k pieces are available, it will return the original value of *clear.* Elements on the input list that are not threshold pieces are ignored. If k pieces are not available, *Threshold-Recover* returns *'Failed.*

4.1.3 Checksums

A *checksum function* maps arbitrary input values into a comparatively small set of output values, such that independent input values have a small probability of being mapped into the same output value. If a c bit checksum is implemented by a cyclic code, it can be proved that the fraction of independent values that have the same checksum is 2^{-c} [Peterson and Weldon 72, p. 229]. Conventional encryption can also be used to create checksums. One way of creating an encryption based checksum is to use a cipher block chain technique [Konheim et al. 80].

Add-Checksum[x: Any / cx: Checksummed-Object]

Add-Checksum returns a copy of *x* and *x*'s checksum.

Check-Checksum[cx: Checksummed-Object / x: Any]

Check-Checksum checks the checksum of *cx*, and if the checksum is correct, it returns the value of *x* it holds. If the checksum of *cx* is incorrect, *Check-Checksum* returns *'Failed.*

4.1.4 Indirection

System-wide indirection is assumed in the sense that an indirect reference can be created on one secure processor with *Create-Indirect*, passed to another secure processor, and still function properly.

Create-Indirect[x: Any / ix: Indirect]

Create-Indirect creates an indirect entry, places *x* in it, and returns an indirect reference to the new entry.

Normalize[ix: Any / x: Any]

Normalize will look up the value of *ix* if it is an indirect reference, otherwise it returns *ix.*

Change-Indirect[ix: Indirect, x: Any]

Change-Indirect changes the indirect entry pointed to by *ix* to contain *x.* If *x* is NIL the indirect entry is deleted.

4.1.5 Unique Identifiers

A unique identifier is a value that is distinct from every other unique identifier. We will assume for simplicity that unique identifiers are system generated nonsensical bit strings, unique over the system and time.

GetUniqueID[/id: UniqueID]

GetUniqueID returns a unique identifier.

4.2 Basic Algorithm

We present the basic algorithm by first outlining the principles of the mechanism and then presenting detailed descriptions of the major functions.

The protection mechanism operates by arranging that precisely enough information is included in a sealed object to allow an appropriate set of keys to unseal it. We will first consider base keys and then show how this is arranged for derived keys.

Base keys are directly implemented by cryptography. If an object is sealed with a base key, then *Seal* encrypts the object with the key. To provide the authentication property, a checksum is added to the object before it is encrypted. *Unseal* uses the checksum to tell if it has a legitimate sealed object. For example, assume that *kb* is a base key. *Seal[x, kb]* is transformed to:

Encrypt[Add-Checksum[x], kb].

Base keys are assigned unique identifiers which are included in the result of an *Encrypt* operation. Thus, it is easy for *Unseal* to determine which key can be used to recover the contents of a sealed object.

A derived key consists of two fields: *key* and *opener*. When an object is sealed with a derived key *dk*, it is sealed with *dk.key*, and *dk.opener* is carried along in the sealed object. *Seal[x, dk]* is transformed to:

[Seal[x, dk.key], dk.opener].

The central idea is that the opener of a derived key is carefully constructed so that if $S >> dk$, then it is possible to compute a key that unseals *dk.key* from *dk.opener* and *S*.

Figure 5 gives several examples of sealed objects. The first example shows what happens when an object is sealed with a base key. The object is encrypted (as suggested by the heavy box), and marked with the key that can be used to decrypt the object. Although it is not shown in the figure, the object in the box includes its checksum for authentication. The next three examples show how derived keys work. It should be clear from the illustration how the opener expresses the protection structure that corresponds to its *Seal* statement.

The following functions implement the protection mechanism. Figure 6 is a diagram of the dependency relationships between the functions, which the reader may find helpful.

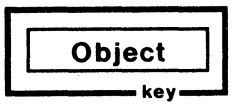
Simple base keys are implemented by conventional cryptography. A simple base key consists of a unique identifier and an encryption key.

```
Simple-Key ← Record[id: UniqueID, key: Byte-Array];

Create-Base-Key[/k: Key] ← Prog[ [];
        k ← Create[Simple-Key];
        k.key ← Create-Conventional-Key[];
        k.id ← GetUniqueID[];
        Return[k];
        ];
```

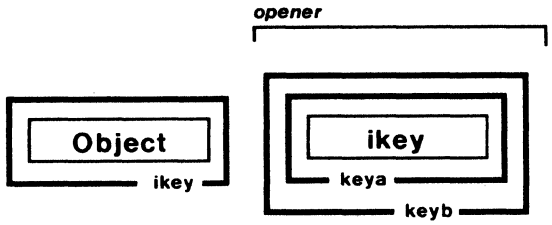
Key pairs are implemented by public-key cryptography. Like a conventional key, a *Key-Pair-Half* has fields for an identifier and a key. A *Key-Pair-Half* also includes the identifier of the key that will decrypt it.

```
Key-Pair-Half ← Record[id: UniqueID, decryptedBy: UniqueID, key: Byte-Array];

Create-Key-Pair[/ kp: Key-Pair] ← Prog[ [pkp: PK-Pair];
        kp ← Create[Key-Pair];
        kp.keya ← Create[Key-Pair-Half];
        kp.keyb ← Create[Key-Pair-Half];
        pkp ← Create-PK-Pair[];
        -- set up keya
        kp.keya.id ← GetUniqueID[];
        kp.keya.decryptedBy ← GetUniqueID[];
        kp.keya.key ← pkp.keya;
        -- set up keyb
        kp.keyb.id ← kp.keya.decryptedBy;
```

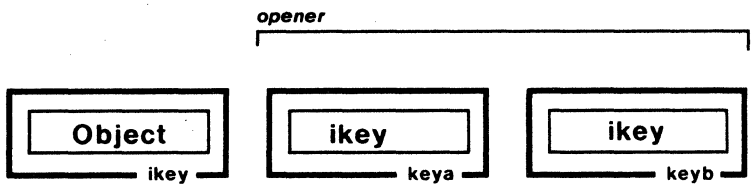
Seal[Object, key]



Seal[Object, Key-And[keya, keyb]]



Seal[Object, Key-Or[keya, keyb]]



Seal[Object, Key-And[keya, Key-Or[keyb, keyc]]]

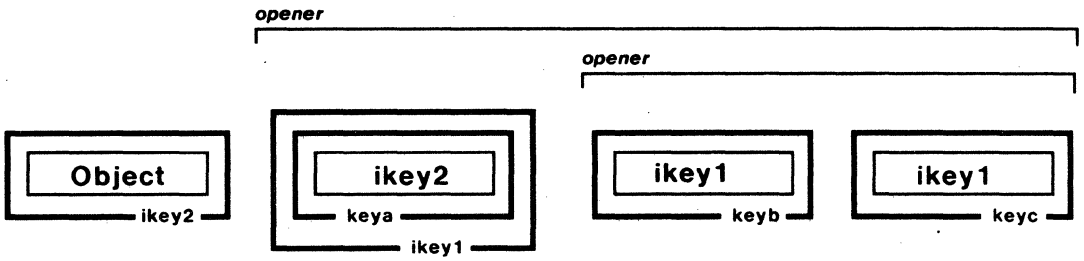
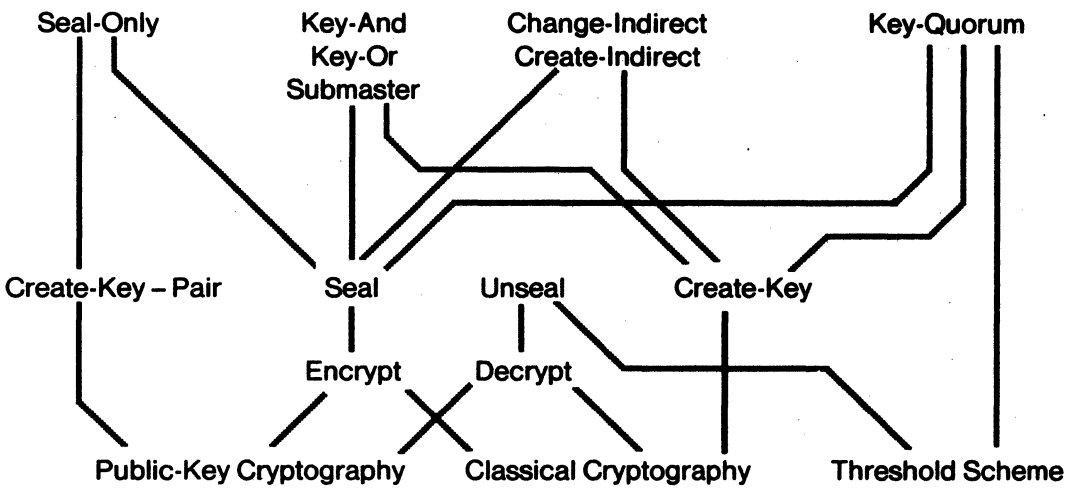


Figure 5. Example Sealed Objects



Protection System Internal Structure

Figure 6

```
        kp.keyb.decryptedBy ← kp.keya.id;
        kp.keyb.key ← pkp.keyb;
        Return[kp];
        ];
```

Before we discuss *Seal* and *Unseal*, let us introduce two cryptographic functions that work with base keys. The *Encrypt* function encrypts an object with a base key, and includes in its result the unique identifier of the key that will decrypt the resulting ciphertext. The *Decrypt* function takes an object that has been encrypted and attempts to decrypt it with the aid of a set of keys.

```
Ciphertext ← Record[decryptedBy: UniqueID, text: Byte-Array];

Encrypt[clear: Any, k: Key / cipher: Ciphertext] ← Prog[ [];
        cipher ← Create[Ciphertext];
        -- encrypt clear with k, producing cipher
        IF Is[k, Key-Pair-Half] THEN [
                cipher.decryptedBy ← k.decryptedBy;
                cipher.text ← PK-Encrypt[Encode[clear], k.key];
                ]
        ELSE [
                cipher.decryptedBy ← k.id;
                cipher.text ← Conventional-Encrypt[Encode[clear], k.key];
                ];
        Return[cipher];
        ];

Decrypt[cipher: Ciphertext, ks: List[Key] / clear: Any] ← Prog[
        [k: Key, rk: Key];
        -- decrypt cipher with one of the keys in ks
        clear ← 'Failed;
        FOR k IN ks DO [
                -- If a derived key, get the base key hidden inside.
                rk ← IF Is[k, Derived-Key] THEN k.key ELSE k;
                -- see if we have a match
                IF Is[rk, Simple-Key] THEN [
                        IF cipher.decryptedBy = rk.id THEN
                          clear ← Decode[Conventional-Decrypt[cipher.text, rk.key]];
                        ];
                IF Is[rk, Key-Pair-Half] THEN [
                        IF cipher.decryptedBy = rk.id THEN
                          clear ← Decode[PK-Decrypt[cipher.text, rk.key]];
                        ];
                ];
        Return[clear];
        ];
```

Seal works in the following way. It first tests the type of key that it has been passed. If it is a base key then *Encrypt* is used to encrypt the object and a checksum of the object. The checksum is used by *Unseal* to authenticate the sealed object. If *Seal* is passed a derived key it uses the base

key contained in the derived key to seal the object, and it includes the opener of the derived key in its result.

```
Derived-Key ← Record[key: Key, opener: List[Sealed-Object]]

Sealed-Object ← Record[cipher: Ciphertext, opener: Any];

Seal[x: Any, k: Key / sealed-x: Sealed-Object] ← Prog[ [];
        IF Null[k] THEN Return[x];
        sealed-x ← Create[Sealed-Object];
        IF Is[k, Derived-Key] THEN [
                -- Include the derived key's opener in the sealed object.
                sealed-x.cipher ← Seal[x, k.key];
                sealed-x.opener ← k.opener;
                Return[sealed-x];
                ];
        -- Add a checksum for authentication.
        sealed-x.cipher ← Encrypt[Add-Checksum[x], k];
        sealed-x.opener ← NIL;
        Return[sealed-x];
        ];
```

Unseal reverses the sealing process. *Unseal* first checks to see what type of key was used to seal an object. If the object was sealed with a base key, *Unseal* uses *Decrypt* to recover the object's contents. The result of *Decrypt* is authenticated by ensuring that the object's checksum is correct. If the object was sealed with a derived key, the opener is unsealed, and the resulting key set is used to unseal the object. *Append* is a Lisp operation that appends one list to another.

```
Unseal[sealed-x: Sealed-Object, ks: List[Key] / x: Any] ← Prog[
        [opener: List[Key]; tp: Threshold-Pieces];
        -- Ensure that the object was in fact sealed.
        IF Not[Is[sealed-x, Sealed-Object]] THEN Return['Failed];
        IF Null[sealed-x.opener] THEN [
                -- object was sealed with a base key
                Return[Check-Checksum[Decrypt[sealed-x.cipher, ks]]];
                ];
        -- The object was sealed with a derived key.
        opener ← Normalize[sealed-x.opener];
        IF Is[opener, Threshold-Pieces] THEN [
                -- The opener is a list of sealed key pieces.
                tp ← Create[Threshold-Pieces];
                tp.pieces ← RUnseal-List[opener.pieces, ks];
                opener ← List[Decode[Threshold-Recover[tp]]];
                ]
        ELSE
                -- The opener is a list of sealed keys.
                opener ← RUnseal-List[opener, ks];
        Return[Unseal[sealed-x.cipher, Append[opener, ks]]];
        ];
```

The derived key *dk* that *Key-And* creates is simple in concept. The opener of *Key-And* can be unsealed with a key set *S* only if *S* >> *ka* and *S* >> *kb*. *ka* and *kb* must not be a key pair. This is because with the implementation of public-key cryptography proposed by [Rivest et al. 78] encrypting with one and then the other is equivalent to encrypting and then decrypting. This restriction could be eliminated at the cost of a small increase in the size of the sealed object by changing the opener to be *List[Seal[Seal[dk.key, ka], Submaster[kb]]]*.

```
Key-And[ka: Key, kb: Key / dk: Key] ← Prog[ [];
        dk ← Create[Derived-Key];
        dk.key ← Create-Base-Key[];
        dk.opener ← List[Seal[Seal[dk.key, ka], kb]]];
        Return[dk];
        ];
```

Key-Or is implemented in the same manner as *Key-And*. One of the elements of the opener of *Key-Or* can be unsealed with a key set S only if S >> *ka* or S >> *kb*.

```
Key-Or[ka: Key, kb: Key / dk: Key] ← Prog[ [];
        dk ← Create[Derived-Key];
        dk.key ← Create-Base-Key[];
        dk.opener ← List[Seal[dk.key, ka], Seal[dk.key, kb]];
        Return[dk];
        ];
```

Key-Quorum creates a new base key and then splits the key into *n* pieces, where *n* is the length of the input key list. Things are arranged so that any combination of *q* of these pieces can be used to reconstruct the new base key. The *n* parts of the base key are then sealed with the *n* input keys, creating an opener that will yield the base key when a key set is available that unseals *q* or more distinct input keys. *Seal-List* is a function that when passed two lists [*a1* ... *an*] [*k1* ... *kn*] returns [*Seal[a1, k1]* ... *Seal[an, kn]*].

```
Key-Quorum[q: Integer, kl: List[Key] / dk: Key] ← Prog[ [];
        dk ← Create[Derived-Key];
        dk.key ← Create-Base-Key[];
        -- Create the proper number of key pieces.
        dk.opener ← Threshold-Split[Encode[dk.key], Length[kl], q];
        -- Seal the pieces with the elements of kl.
        dk.opener.pieces ← Seal-List[dk.opener.pieces, kl];
        Return[dk];
        ];
```

Submaster is implemented by creating a new base key, and including the key in an opener sealed with *k*.

```
Submaster[k: Key / dk: Key] ← Prog[ [];
        dk ← Create[Derived-Key];
        dk.key ← Create-Base-Key[];
        dk.opener ← List[Seal[dk.key, k]];
        Return[dk];
        ];
```

Seal-Only ensures that only sets that unseal k will unseal dk. This derived key is set up to encrypt objects with one half of a key pair, and to hide the other half in an opener. The opener can only be unsealed with a key set S if $S \gg k$.

```
Seal-Only[k: Key / dk: Key] ← Prog[ [kp: Key-Pair];
        kp ← Create-Key-Pair[];
        dk ← Create[Derived-Key];
        dk.key ← kp.keya;
        dk.opener ← List[Seal[kp.keyb, k]];
        Return[dk];
        ];
```

Create-Indirect-Key creates a key that can be changed. This is accomplished by creating a derived key with an indirect opener. By changing the indirect opener, one can change the keys that unseal the indirect key.

```
Create-Indirect-Key[k: Key / dk: Key] ← Prog[ [];
        k ← Submaster[k];
        -- create indirect key
        dk ← Create[Derived-Key];
        dk.key ← k.key;
        dk.opener ← Create-Indirect[k.opener];
        Return[dk];
        ];

Change-Indirect-Key[dk: Key, nk: Key] ← Prog[ [];
        -- if nk is NIL, delete key
        IF Null[nk] THEN
                Change-Indirect[dk.opener, NIL];
        ELSE
        -- replace opener with new one
                Change-Indirect[dk.opener, List[Seal[dk.key, nk]]];
        Return[];
        ];
```

4.3 Strength

Assuming perfect cryptography, Gifford [Gifford 81] demonstrates that cryptographic sealing always provides the secrecy and authentication properties. Of course, cryptography is not perfect. Often, breaking a practical cryptographic system is a matter of economics [Diffie and Hellman 77]. However, if some guidelines are followed, the susceptibility of our protection system to cryptanalysis can be reduced.

The cryptosystems that are used must be secure against a known-cleartext attack. The checksums that are included in sealed objects and the implementation of *Key-And* increase the probability that an intruder will be able to use a known-cleartext attack.

It is wise to keep the amount of information protected with a single key to a minimum. This makes it more difficult for an intruder to perform a known-cleartext attack, and it reduces the vulnerability of the system to a single cryptanalytic success. Because keys are inexpensive, a client should use as many keys as are natural for its application.

The strength of the cryptographic system used to protect information should correspond to the length of time the information needs to be kept secret. For example, if certain information is going to be sensitive for twenty years, it would not be wise to protect it with a cryptosystem that can be broken in one year. Another metric is that the value of information protected with a cryptographic system should be considerably less than the cost of an attack on the system. Cryptographic sealing can accommodate the coexistence of a number of cryptosystems that have different key sizes and strengths. Thus, the strength of a cryptosystem can be matched to the sensitivity of the information it protects.

Finally, there is no need to divulge information that might lead to a successful cryptanalytic attack to clients that do not need to know the information. For example, public keys can be protected from clients that do not need them.

5. Practical Considerations

Implementation

We expect that an implementation of cryptographic sealing could be modeled after the program text in the paper. This is because the text itself was adapted from an Interlisp-D prototype of the mechanism. The prototype was not directly useful because it did not perform the necessary cryptographic functions. The intent of the prototype was to understand how to create a simple set of recursive functions to implement cryptographic sealing, and to verify our understanding of the algorithm by trying out some examples on the prototype implementation. The prototype led to a simplification from the original approach.

Key Authentication

A practical system must ensure that external names are mapped into correct internal keys. There are a number of solutions to this problem. Needham and Schroeder [Needham and Schroeder 78] have demonstrated one way of transforming external names into authentic internal keys.

Another approach would be to create a key pair, [keya, keyb]. *keya* is kept secret by the system administration, and <name, public-key> pairs in the system directory are sealed with *keya*. *keyb* is made public, and given to users (perhaps in a face to face meeting). It is possible for a user to tell if an entry in the public directory is authentic by unsealing it with *keyb*.

Revocation

The problem of revoking access to an object once it has been granted is known as revocation [Redell 74]. An indirect key does not provide revocation. Clients can hold on to the contents of indirect openers, and thus continue to access items sealed with the indirect key, regardless of how the opener might be updated. Revocation can be implemented with the help of a simple active protection system [Gifford 81].

Performance

The performance of cryptographic sealing completely depends on the performance of its foundation. High performance VLSI components have lowered the cost of conventional cryptography to a point where it can be ignored for practical purposes. For example, an encryption unit built by the author for the Xerox Dolphin processor can encrypt or decrypt 512 bytes in 332 microseconds. However, the performance of public-key cryptography and threshold schemes may be a significant consideration. The time for a single public-key encryption or decryption is currently measured in fractions of a second. The following refinements are intended to improve the performance of cryptographic sealing when keys from *Create-Key-Pair* or *Key-Quorum* are used:

1. *Unseal* discards keys derived from openers as soon as they are no longer needed. An important optimization would be to remember these keys, and use them to reduce the number of future unseal operations. When such keys were not in use, they could be stored on a user's key ring.

2. When client X is about to seal an object with K, a key-pair half, it creates K',

   K' ← Submaster[K],

   and seals the object with K' instead of K. K' is then stored in a cache in X's secure processor. In the future, if X is going to seal another object with K, it uses K' instead.

   Now imagine that client Y is unsealing the objects that client X has sealed in this manner. If client Y has adopted the previous suggestion, it will only have to perform a single public-key decryption to recover an entire set of sealed objects produced by X. This case commonly arises when one user is granting privileges to another user.

In some cases the authentication property of the protection mechanism may not be required. In such instances the checksum in sealed objects could be eliminated, reducing their size.

6. Comparative Analysis

To understand how cryptographic sealing might be used it is important to understand its advantages and disadvantages when compared with traditional protection mechanisms. The disadvantages of cryptographic sealing are:

1. If integrity or availability guarantees are required, a supplemental active mechanism must be used.
2. A user that holds a privilege can easily grant the privilege to another user by giving him the corresponding key. For example, if a user has a key that authorizes him to access top secret information, the user can give that key to anyone he chooses.
3. The ability to revoke a privilege that has been granted to a user requires a supplemental active protection.
4. User names must be translated into authentic keys.
5. If a client makes extensive use of key-pairs, the cost of the underlying public-key cryptography may be prohibitive.

However, cryptographic sealing provides a number of advantages not found in traditional systems:

1. Absolute privacy is provided in the sense that a user does not have to trust the computer system where his information is stored. Even a system operator or administrator can not gain access to information that he has not been authorized to see. Because there are no locksmiths for the mechanism, however, care must be taken not to lose critical keys.
2. The protection system is not privileged. Specifically, a client is free to modify its copy of the protection system code. Moreover, a client can not increase its privileges in any way by such modification.
3. Physical security of storage devices and the protection mechanism are not required. However, we assume that a client has a secure processor in which it can operate on unsealed objects.
4. It is possible to protect information that is freely distributed (e.g. a physical storage medium that is sold) and information that is broadcast.
5. Cryptographic sealing allows a simple active protection mechanism (such as the guard mechanism outlined in Section 3.2) to implement a variety of protection policies that include integrity and availability guarantees.
6. Cryptographic sealing protects variable size objects. Thus, different objects in a single file can be protected in independent ways.

These advantages make cyptographic sealing well adapted for use in decentralized computer systems.

7. Conclusion

Starting with cryptography, we described a new mechanism for protecting data. Our approach to protection is novel in that the contents of storage can be read by anyone, and yet information is kept secret from those who have not been granted access. One of the results of the mechanism is that absolute privacy is provided. A number of interesting applications of the protection mechanism were discussed, including capabilities, access control lists, and information flow control.

Acknowledgements. I would like to thank Dorothy Denning, Dan Greene, Jim Gray, Butler Lampson, Roger Needham, Susan Owicki, Larry Stewart, Howard Sturgis and the referees for their comments and suggestions.

Appendix

The purpose of this appendix is to describe the extensions we have made to Lisp 1.5. For full details on Lisp 1.5 the reader is advised to consult [McCarthy et al. 62].

Syntax

To improve the readability of source programs a number of minor syntactic extensions have been added to M-Expressions. These extensions have been chosen to make programs easy to read for people who are familiar with a contemporary algebraic language.

Any line that begins with two dashes "--" is ignored. Also, the expression after a colon ":" is ignored. This allows the types of variables to be included as comments in the source text. The expression "Any" means the variable contains an arbitrary expression.

Function definitions are of the form

FunctionName[arg1 ... argn / returns] ← *Expression;*

where *returns* is the return variable of the function.

Records

A record type is created by applying the function Record to a list of the fields that the new record might contain. A record instance is made by applying Create to a record type. For example:

```
Person ← Record[age: Integer, name: Byte-String];
Joe ← Create[Person];
```

The operator to access the fields of a record is period ".". Thus, to set the age of Joe, we could write:

```
Joe.age ← 28;
```

It is also possible to test to see if a variable is an instance of a certain type of record. Is[x, t] returns T if x is a record of type t. Thus, Is[Joe, Person] would return T.

Encode and Decode

Encode converts an object into an array of bytes, and is similar to the Lisp function Print. An object in encoded format may be preserved outside of Lisp for any length of time. Decode converts an object in encoded format back into a form that can be directly manipulated. Decode is similar to the Lisp function Read. Decode[Encode[x]] is equivalent to Copy[x].

References

[Blakley 79] Blakley, G. R., Safeguarding Cryptographic Keys, *Proc. 1979 National Comp. Conf.*, AFIPS Press, pp. 313-317.

[Chaum and Fabry 78] Chaum, D. and Fabry, R., Implementing Capability-Based Protection Using Encryption, Rep. UCB/ERL M78/46, Electronics Research Laboratory, University of CA, Berkeley, July 1978.

[Denning 76] Denning, D. E., A Lattice Model of Secure Information Flow, *Comm. ACM 19*, 5 (May 1976), pp. 236-243.

[Dennis and Van Horn 66] Dennis, J. B., and Van Horn, E. C., Programming Semantics for Multiprogrammed Computations, *Comm. ACM 9*, 3 (March 1966), pp. 143-155.

[Diffie and Hellman 76] Diffie, W., and Hellman, M. E., New Directions in Cryptography, *IEEE Trans. on Inf. Thy. IT-22*, 6 (November 1976), pp. 644-654.

[Diffie and Hellman 77] Diffie, W., and Hellman, M. E., Exhaustive Cryptanalysis of the NBS Data Encryption Standard, *Computer* (June 1977), pp. 74-84.

[Diffie and Hellman 79] Diffie, W., and Hellman, M. E., Privacy and Authentication: An Introduction to Cryptography, *Proc. of the IEEE 67*, 3 (March 1979), pp. 397-427.

[Gifford 81] Gifford, D.K. *Information Storage in a Decentralized Computer System*, Ph.D. Thesis, Stanford University, June, 1981. Also available as report CSL-81-8, Xerox Palo Alto Research Center.

[Gudes 80] Gudes, E., The Design of a Cryptography Based Secure File System, *IEEE Trans. on Soft. Eng. SE-6*, 5 (September 1980), pp. 411-420.

[Konheim et al. 80] Konheim, A.G., et al., The IPS Cryptographic Programs, *IBM Sys. J. 19*, 2 (1980), pp. 253-283.

[Lindsay and Gligor 78] Lindsay, B., and Gligor, V., Migration and Authentication of Protected Objects, *IEEE Trans. Soft. Eng. SE-5*, 6 (November 1979), pp. 607-611.

[McCarthy et al. 62] McCarthy, J., et. al., *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, MA, 1962.

[Morris 73] Morris, J. H., Protection in Programming Languages, *Comm. ACM 16*, 1 (January 1973), pp. 15-21.

[Needham 79] Needham, R. M., Adding Capability Access to Conventional File Servers, *Operating Systems Review 13*, 1 (January, 1979), pp. 3-4.

[Needham and Schroeder 78] Needham, R. M., and Schroeder, M. D., Using Encryption for Authentication in Large Networks of Computers, *Comm. ACM 21*, 12 (December 1978), pp. 993-999.

[Peterson and Weldon 72] Peterson, W. W., Weldon, E. J., *Error-Correcting Codes*, Second Edition, MIT Press, Cambridge, MA, 1972.

[Redell 74] Redell, D. D., Naming and Protection in Extendible Operating Systems, Ph.D. Thesis, University of California, Berkeley, September 1974. Available as TR-140, MIT Laboratory for Computer Science, Cambridge, MA.

[Rivest et al. 78] Rivest, R. L., Shamir, A., and Adleman, L., A Method of Obtaining Digital Signatures and Public-Key Cryptosystems, *Comm. ACM 21*, 2 (February 1978), pp. 120-126.

[Saltzer and Schroeder 79] Saltzer, J. H., and Schroeder, M. D., The Protection of Information in Computer Systems, *Proc. of the IEEE 63*, 9 (September, 1975), pp. 1278-1308.

[Shamir 79] Shamir, A., How to Share a Secret, *Comm. ACM 22*, 11 (November, 1979), pp. 612-613.

[Shannon 49] Shannon, C. E., Communication Theory of Secrecy Systems, *Bell Sys. Tech. Journal 28*, (October 1949), pp. 656-715.

XEROX

Cryptographic Sealing for Information Secrecy and Authentication

David K. Gifford

CSL-82-1